NAS Technical Report
NAS-05-016
December 2005

Creation of a Multilevel Parallel Version of CFL3D

Robert J. Bergeron

bergeron@nas.nasa.gov

NAS Systems Division
NASA Ames Research Center
Mail Stop 258-5
Moffett Field, CA 94035-1000

Abstract

Multilevel parallel versions of CFL3D have been created and their performance has been evaluated for a complete aircraft problem on NAS platforms. CFL3D is a three-dimensional multi-block Navier-Stokes flow solver for structured grids and the official version of the code employs MPI to execute calculations in parallel over the discrete geometrical regions. A second version of the code employing the MLP library was created to compare its performance to that of the MPI library. The major modification to CFL3D, involving the aggressive exploitation of parallelism in the main computational loop, was added to both versions of the code using the OpenMP library. Elimination of the output routine and reassembly of the output after execution increased the effectiveness of all code improvements.
On the SGI 3000 architecture, the overall improvements have produced a speedup of 8.5 in the most time-consuming iterations and implementation on the SGI Altix has further increased this speedup by a factor of 5.5. The MLP library demonstrates a performance somewhat superior to the MPI library because of its superior load balancing capabilities.

1. Introduction

Application programs sometimes require modification to exploit the computational power of modern supercomputers. To add a significant amount of value, such modification efforts should target those applications used extensively and then apply the proper set of modifications. This report will discuss the application of multilevel parallelism to a major aeronautics design code. This section describes the aeronautics code and outlines the efforts made to exploit additional parallelism in this code.
CFL3D is a Reynolds-Averaged Navier-Stokes thin layer time-dependent solver for structured grids[1].The code has been employed in a wide variety of applications, ranging from low subsonic turbomachinery simulations to hypersonic National AeroSpace Plane design.Recent success with another computational fluid dynamics code, OVERFLOW[2], motivated a request to NAS to investigate the application of multilevel parallelism to CFL3D.
CFL3D employs upwind differencing and flux-vector splitting in a finite-volume solution framework.This code employs mesh sequencing, multigrid and local timestepping to accelerate convergence to the steady state. Time-accurate calculations employ implicit approximate factorization and subiterations to improve temporal accuracy. The code partitions complex geometries into discrete regions and solves the governing equations in parallel over these regions using the MPI library. This code employs a master / worker approach in which the worker visits regions whose data is local to the worker process. MPI send / receive pairs transmit convergence-related data to the master for evaluation and printout.The CFL3D code uses the term grid to refer to the set of points comprising the physical flow-

field and block to describe the fine grid and various sub-grids.
The code consists of three parts: a preface, a compute section, and the wrap-up. The preface reads the initial and boundary conditions, performs the grid setup, reads the restart, initializes quantities necessary to the problem such as the minimum distance to the nearest viscous wall, and computes cell-interface areas.
The compute section manages several types of boundary conditions, and executes the main loop which visits all the blocks. This loop evaluates the turbulence and the flux contributions to the right-hand side, and advances the solution in time. The loop executes the routines which drive the computationally-intense sections of CFL3D, and exposure of OpenMP parallelism in these routines was a key to achieving scalable performance.
The wrap-up writes the restart, calculates turbulent flow quantities at solid surfaces, and outputs various data for plotting or printing.
A technique for combining high-level decompositions with fine-grained parallelism, MultiLevel Parallelism (MLP) has achieved some success recently on SGI platforms [2]. Applying both coarse-grain and fine-grain levels of parallelism to a single code is not new. The Cray shared-memory platforms provided this technique in 1989 through the combination of macrotasking and microtasking. The use of these utilities in combination was not widespread, due in part to the large memory requirements of problems which would justify their use and also due in part to the somewhat arcane problems encountered in their use.
The great advantage of the MLP approach is the ease in expressing the parallelism and obtaining scalable performance. This approach employs the UNIX '*fork*' command to spawn a set of coarse-grain processes at the user's discretion. MLP allows insertion of OpenMP [3] loop-level directives into computationally intense regions of the code because each MLP process is an independent high-level process. Communication of shared memory data occurs through FORTRAN loads and stores, and allocation of the shared memory requires simple FORTRAN statements. The MLP library has undergone several revisions to increase its performance; in particular, the addition of a set-and-test functionality has allowed MLP to add an asynchronous feature to its repertoire and this feature strongly influences CFL3D performance. The simplicity and reliability of the OpenMP library promotes a second level of parallelism for the MPI and MLP approaches. Efficient employment of OpenMP involves the exposure of large (spanning subroutines) secondary parallel constructions to reduce the load imbalances generated especially by the high-level decomposition of CFD geometries.
The CFL3D versions discussed herein have executed on NAS machines with somewhat different architectures, and this report uses the term node to refer to a group of CPUs sharing a common memory. Thus the NAS platform Lomax has 128 nodes with 4 CPUs sharing each node's memory for a total of 512 CPUs and the NAS platform Columbia consists of multiple 256-node cabinets with 2 CPUs sharing a node's memory.
The MLP library uses shared memory as a common location for data to be accessed by multiple processes, and the user must explicitly allocate such shared memory based on an understanding of what data needs to be shared. Sections 1 and 2 will clarify the data required to reside in shared memory, and Section 3 discusses some aspects of shared memory optimization. CFL3D uses the MPI library to transmit data from one process to another; the MLP version uses simple loads and stores to do the same transfer and Section 4 discusses this modification to the MPI version. The ASCII output routine for many traditional single processor codes is frequently a performance bottleneck and becomes more so as such codes are converted to parallelism; Section 5 explains the removal of this bottleneck for both versions of CFL3D.The second level of parallelism involves the use of additional processes (termed threads to indicate their limited nature, both inmemory and existence) in computationally intense regions to reduce wall-clock times and Section 6 describes the exposure of this additional parallelism. For both MPI and MLP versions, scalable performance, i.e.,a wall-clock time which continues to decrease in a proportional fashion with the addition of more processors, requires proper memory placement of the OpenMP threads to processors. Section 7 discusses this thread / processor placement. Section 8 compares MPI and MLP performance on a large test problem and Section 9 presents some conclusions.

2. Sizing of MLP Shared Memory

The shared memory serves as a common storage area to allow the MLP loads and stores to replace the MPI_Send and MPI_Recv calls.The sizing of shared memory involved several iterations, based on the experience gained from the test problems.The size of the shared memory must be set before creating (forking) the high-level parallel processes. The MPI version of CFL3D invokes subroutine SIZER before execution of the solution algorithm to set the dimensions of the key arrays, and the MLP version also used SIZER to establish the dimensions of the shared memory array.

Typically, boundary conditions and restart requirements set the size of shared memory because the former involves exchanging neighboring grid partial solutions and the latter requires the workers to send their entire solution to the host for transfer to the restart file. Since all the MLP processors fill the shared memory array(s) simultaneously, such array(s) must be properly sized, i.e., according to the problem input specifications.

Experience with CFL3D problems indicated that boundary conditions determined the shared memory size for some problems and the restart requirements determined the shared memory size for others. The SIZER subroutine uses the actual loops from the boundary conditions to determine the size of the arrays which must be sent from one process to another. Similarly, SIZER uses the actual loops from the restart routines to determine the size of the arrays which must be sent from the workers to the master. Thus, SIZER establishes the maximum amount of MLP shared memory required for a process and for the entire problem. The starting addresses of processor data determined by SIZER are placed into arrays, and since SIZER executes before MLP_FORKIT spawn its high-level processes, all MLP processors inherit these address arrays.

The shared memory array containing floating point data will be referred to as wkS; this array serves so many purposes throughout the calculation that CFL3D requires only one other shared memory array, iawkS, which contains integer addresses in wkS of the boundary condition data.

3. MLP Shared Memory Usage

The operating system allocates the shared memory segments from within the standard memory allocated to the CPUs executing the MLP program Since such memory is a limited resource, shared memory usage should be monitored and optimized. Both the monitoring and the optimizing of memory were required to construct a robust MLP version of CFL3D.

The total memory used by the program consists of local memory and shared memory, and if this total exceeds the memory of the CPUs requested by the job, the program will die. CFL3D lays out the worker processes sequentially by memory requirements, beginning with the largest, and since the MLP processes share node memory, several large processes executing on the same node can oversubscribe the node memory. The operating system may request the extra memory from the next available node to satisfy the oversubscription, and the effect will be to slow code performance since memory access will be over the network. Fortunately, the NAS SGI Origins allowed the development of a local code, *fmem*, by Bob Ciotti for nodewise tracking of memory used by a job, and this tool aided greatly in the development of the MLP version of CFL3D. *Fmem* provides a snapshot of free memory at a rate selected by the user, and combined with *sleep* calls inserted into the code, enables the user to easily identify the source of any memory oversubscription.

CFL3D has two features which impact memory usage and which presented problems for the MLP version. After assigning the master process to the first available CPU, CFL3D assigns the process with the largest grid dimensions to the next available CPU. The code continues to place processes onto CPUs according to memory requirements. The Lomax architecture requires that 4 CPUs share a single node's memory, and the CFL3D scheduling algorithm will place three processes with the largest memory requirements on the first node, thereby inadvertently challenging the nodal memory capacity.

Additionally, the MPI CFL3D memory arrangement assigns the same amount of memory to each high-level process, the total memory being composed of the q-array (array containing mass, momentum, and energy) and the scratch array. After computing the maximum process scratch memory requirements, the POINTER subroutine simply adds memory to the scratch arrays belonging to the other processes, so that all processes have the same memory requirements. This approach works fine when the code executes a

single level of MPI parallelism, perhaps on a cluster of workstations. The approach runs into memory oversubscription and subsequent performance problems when running multi-level parallelism on a shared memory computer.The MLP solution to the scratch size problem was to use the data available in subroutine POINTER to return the proper scratch array size and this approach produced a considerable reduction in memory usage.
To solve the shared memory sizing problem, MLP always packs the shared memory array in one-dimensional fashion to avoid any extra usage arising from 2 or 3 dimensional addressing. Furthermore, examination of the CFL3D data flow indicated that only a single shared memory array was necessary. After modifications reducing the scratch array size and shared memory usage were implemented, the MLP version experienced no performance problems from excess memory requirements.

4. MLP Data Transfer

The typical MLP conversion of an MPI parallel code begins by replacing MPI library send / recv pairs with a Fortran load-barrier-store sequence, in which the data to be transferred is loaded into shared memory, an MLP barrier is then set to prevent the processors from executing until the load has completed, and finally the shared memory data is stored into local memory. This approach is sometimes described as a “ping-pong” approach.The current MLP version has replaced about 80 of the 300 MPI calls with MLP loads and stores; additional replacements will be straightforward when problems exercising these parts of the code are executed.
The CFL3D boundary conditions illustrate the additional applicability of the MLP data transfer technique to MPI asynchronous message-passing (Isend / Irecv). BC_BLKINT employs asynchronous message-passing to exchange block interface boundary conditions for grids sharing a continuous common interface and BC_PATCH does the same for grids sharing a noncontinuous common interface. In the asynchronous treatment, each MPI worker process posts a series of messages, essentially to a queue containing the number of the receiving process and the locations of the data. After posting the receives and sends and executing an MPI_WAITALL, the boundary routines execute a loop over the number of messages until the loop has processed all messages.
The MLP version simulates the message-passing in the SIZER subroutine to obtain the amount of memory needed by each process in the wkS array. The boundary routines also contain an additional simulation to place the index of the location for each message in the wkS array into another shared memory array, iawkS. The boundary routines then execute in a fast ping-pong fashion in which each high-level process loads its transmitted data into the wkS array, waits for all such loads to complete using a single MLP barrier call, and then stores the data it received from the wkS array into a local memory array and proceeds.
CFL3D also supports overlapping and embedded grid boundary conditions, but the current test problems do not request these types and so these routines currently lack the required MLP modifications.

5. High-Level Processor Optimization

Two CFL3D problems requiring some high-level processor optimization of the code surfaced during the creation of the multilevel parallel versions, one was specific to the MLP version and the second applied to both versions.
In subroutine RESP (Residual Print), the worker processes send the data required to monitor density and turbulence residuals, and force / moment coefficients to the master process for evaluation and printout. Since the amount of data transmitted to the master is small, MPI performs such transfers asynchronously (and very fast) with the MPI library accounting for the extra memory via system buffers. The MLP code with its new set-and-test routine was able to replace its ping-pong treatment and also perform such small data transfers asynchronously; however, the MLP code must explicitly account for the extra memory required to perform such transfers. The new MLP library containing the set-and-test routine is currently available only on the Altix machines; no effort was made to port it to the SGI 3000 architecture since NAS will use the Altix machines in the future.
The second optimization involved formatted CFL3D output. Timing studies indicated that the CFL3D test problems spend most of their time in the output routine, WRIT_BUF, because once a worker has

prepared data for the master, the worker must wait until the master process calls for it. The workers send to the master, on a blockwise basis, ASCII data detailing the timestep calculations, the boundary conditions, and the general progress of the calculation.In contrast to the RESP subroutine discussed above, the amount of data transmitted in WRIT_BUF is large and MPI does not readily allow asynchronous processing of such requests.Initially, MLP development followed the MPI approach, transmitting the local convergence data using shared memory, but the resulting modification produced no significant reduction in wall-clock time due to the serial nature of the output processing.
Following a technique used by Jim Taft in the OVERFLOW code[2], MLP worker processes now simply write their output to a local file and return to the calculation. CFL3D has about 30 separate calls to WRIT_BUF and master and worker processes now write the number of output call on their respective files. Upon completion of the problem, the master's file contains its global output plus these call flags indicating the type of output which would have been written had the worker-to-master transmittal occurred. The worker files contain an output flag written before each set of local convergence data. Successful placing of the output flags onto the master's file required some experimentation due to the tendency for records to be compressed or overwritten. When the problem completes, a simple Fortran code reads the master file and uses its output flags to copy the proper worker data from the worker files onto the master file. Reassembly takes 1 or 2 seconds, and for the current five test problems, the reassembled output files agree with the MPI output file, apart from a few blank lines and spaces.
Removal of the serial output processing in CFL3D, its consequent reduction in wall-clock time and the relative ease of this task should make this approach a general prescription for all parallel codes. At this stage in the development of parallel codes, no code should spend a significant fraction of time processing ASCII output.

6. Second-level Parallelism with OpenMP

The multilevel parallel approach enjoying some success on NAS application codes does not rely on compiler-driven or tool-driven exposure of parallelism.Instead, this approach seeks an understanding of how the code operates on the key data arrays and then exposes parallelism at a high level to give the processors large chunks of work. Identifying the subroutines or code loops which can act as the parallel drivers for this high-level parallelism sometimes presents challenges.The CFL3D manual[1] does indeed identify all such driver subroutines; however, exposing the parallelism contained therein required some effort. Codes lacking such well-written manuals will require an execution flowtrace to help identify potential OpenMP driver subroutines. Such a flowtrace can be constructed by placing print statements at the beginning of each subroutine and subsequently reducing the output to manageable levels.
For each multigrid level, the MPI or MLP process executing the main computational loop in MGBLK visits the blocks belonging to the level whose data is local to that processor. If the data is local to the processor, each process calls subroutines RESID to evaluate the right hand side (fluxes) and AF3F to advance the solution in time. RESID and AF3F contain loops over grid planes in various J-K-I directions which evaluate fluxes. These routines control the two major chunks of computational work in the main loop.
The original MPI version chunks these DO-loops, i.e., the code divides the number of planes by a convenient number, 64 for example, to improve performance and then makes 1 or 2 passes through these loops. The small number of loop iterations does not allow extra OpenMP threads to assist in the execution of these loops to reduce the wall-clock time
The OpenMP version modifies the routines RESID and AF3F to remove the chunking from these loops, allowing extra threads to participate evenly in their execution. Identification of RESID and AF3F as the OpenMP driver subroutines and the subsequent removal of the loop chunking was the **key** to reducing wall-clock time with the second level of parallelism. This action illustrates aggressive user intervention in parallel regions.Compiler or parallel tool creation of the parallel regions is inefficient; for example, it is highly unlikely that a parallel tool would even be able to recognize the importance of such loops or to create parallel regions which span subroutines effectively. The use of OpenMP in the main loop is the only place extra OpenMP threads assist the CFL3D computation.
The multilevel parallel versions call the OpenMP library before the main loop to make available extra processors to assist in the execution of loops in the subroutines called by RESID and AF3F. Two

complications arise from OpenMP usage:
-OpenMP threads share variables in the COMMON blocks, so only the master OpenMP thread may update these variables and this restriction applies to all subroutines in the span of control of the two driver subroutines.
-Memory access (loading and access of arrays) by multiple OpenMP threads requires careful management. In subroutine RESID, the work array, wk, provides both thread private and thread shared scratch space. When executing under the OpenMP library with the multilevel parallelism, the OpenMP threads need separate wk arrays and a special subroutine, DMGBLK0, instantiates the wk array as wk(1,maxthr) where maxthr is the number of OpenMP threads for this multilevel processor. For thread-private memory storage, the code addresses wk as wk(1,iam) where iam is the thread number, and for thread-shared memory storage, the code addresses wk as wk(1,1).
Verification of the MLP library and OpenMP library modifications is carried out by examining the q-arrays (arrays containing the density, momenta, and pressure) after the main loop. For all current test problems, sums of the q-arrays agree to 8 digits after the decimal point with the checksum computed with the released MPI version. In addition, both multilevel versions have executed correctly under OpenMP all turbulence models currently allowed in CFL3D.

7. OpenMP Processor Memory Layout

The OpenMP threads assist the high-level processes by participating in the execution of certain driver subroutines and the DO LOOPS comprising these routines. Before the OpenMP threads begin their work, the OpenMP library copies the required data, typically OpenMP shared arrays, from the high-level process to the processor(s) designated by the operating system as its OpenMP processor(s). Although such copying is transparent to the user, it is overhead.If the MLP process and its OpenMP thread(s) share a common memory, the copy overhead can be very small If the high-level process and its OpenMP thread(s) do not share a common memory, the copy overhead may be very large and may adversely affect performance.
The MPI standard does not prescribe the placement of OpenMP thread(s) relative to their high-level process. Such arrangements are left to the operating system, with UNIX environment variables typically granting the high-level process only a uniform number of OpenMP thread(s) and limited leeway regarding their placement. Such restrictions do not allow efficient load-balancing of an aircraft simulation.
In contrast, the MLP library call, MLP_FORKIT, enables efficient sharing of local memory by allowing the user to specify the MLP processes and the OpenMP threads sharing a common memory. The user specifies the number of OpenMP threads to be used by the MLP processor. If 4 processors share the local memory (4 processors constitute a node), the MLP can ask for 1, 2, or 3 or, for large blocks, even 7 OpenMP threads, without incurring a copy penalty.
The MLP version of CFL3D reads an input file specifying the OpenMP arrangement and thus the process memory layout. The code passes this data to the MLP_FORKIT routine. This routine assigns MLP processes to the CPUs in a sequential fashion, reserving the number of CPUs requested for OpenMP threads.
Table 1 illustrates the different grid sizes and the OpenMP layout for the Complete Aircraft simulation discussed in the next section. The Table describes a 96-grid,72-node, 288-CPU simulation run on Lomax and shows the distribution of OpenMP threads required to balance the computational load.

Table 1: Complete Aircraft MLP/OpenMP Processor Layout on Lomax

MLP Process	OpenMP Threads	Node	Type	Comment
01	1	1	No Grid	master-1 CPU
02	7	1&2	Large Grid	spans 2 nodes
03	4	3	Large Grid	spans 1 node

Table 1: Complete Aircraft MLP/OpenMP Processor Layout on Lomax

MLP Process	OpenMP Threads	Node	Type	Comment
04	4	4	Large Grid	spans 1 node
...	...	...	Large Grid	spans 1 node
57	4	57	Large Grid	spans 1 node
58	2	58	Medium Grid	spans 1/2 node
...	...	...	Medium Grid	spans 1/2 node
69	2	63	Medium Grid	spans 1/2 node
70	1	64	Small Grid	spans 1/4 node
...	...	...	Small Grid	spans 1/4 node
74	1	65	Small Grid	spans 1/4 node
75	2	65	Medium Grid	spans 1/2 node
...	...	...	Medium Grid	spans 1/2 node
84	2	69	Medium Grid	spans 1/2 node
85	1	70	Small Grid	spans 1/4 node
...	...	...	Small Grid	spans 1/4 node
96	1	72	Small Grid	spans 1/4 node

8. Performance

The multilevel parallel CFL3D versions have executed 5 different problems correctly, and the best example of code performance is a 96-grid, 25-million-point problem involving a complete aircraft. As shown above, this problem consists of a variety of small, medium, and large grids corresponding to typical aircraft sections. The CFL3D simulation calls for 3 levels of multigrid consisting of the fine grid and two coarse grids. In a multigrid code, most of the wallclock time will be spent visiting the fine grids, and as the code makes hundreds or thousands of visits to converge the solution, the wallclock time will largely consist of the time spent in the fine grid, and multilevel parallelism should display the greatest improvements in these fine grids. CFL3D will visit all the fine grids in the first pass of the main computational loop, but subsequent passes may skip those grids which have converged. The timings below are the average wallclock times for the first pass through the fine grid and the average wallclock times for the time spent in all passes of the fine grid. Timings are reported for execution on two NAS platforms, the SGI Lomax and the newer SGI Columbia system, and for execution with two high-level process counts. This paper characterizes the performance improvements by using the term speedup to refer to the ratio of the wallclock time of the base case to the wallclock time of the case under consideration.

In the 32-process case, each process treats 3 physical grids, and the load balance occurs largely through the CFL3D grid placement on the high-level processors. In the 96-process case, each processor treats a single grid and the OpenMP attempts to load balance come into play. The MPI/OpenMP CFL3D version requires a special memory placement feature to operate efficiently on the Origin and Altix platforms. The environment variable NUMTHREADS allows the operating system to space out the MPI processes to

accomodate the OpenMP threads associated with each MPI process and to insure that the OpenMP threads remain close to the parent MPI process.
All timings were obtained by normal execution in the NAS PBS workload.

Lomax

The Lomax machine is a SGI 3000 architecture consisting of 512 1.2 GFLOP CPUs, each with 512 Mbyte of memory and an 8 Mbyte cache. Table 2 compares CFL3D performance on Lomax, illustrating the effects of the fast I/O and multi-level parallelism.The Lomax timings were made with revision 3.0 of the MLP library; pairwise synchronization of MLP processes, e.g., master and worker in RESP, involved an mlp_barrier call.

Table 2: CFL3D Complete Aircraft Timings on SGI 3000 Lomax

Type	Process Count	OpenMP Threads	Firstcycle	Finegrid	Comment
MPI	32	0	88.9	48.1	Original WRIT_BUF
MPI	32	0	29.4	14.7	Fast WRIT_BUF
MLP	32	0	64.2	32.1	Fast WRIT_BUF
MPI	32	32	18.7	9.3	NUMTHREADS=2
MLP	32	32	39.1	20.3	OpenMP load balanced
MPI	96	0	69.7	36.9	Original WRIT_BUF
MPI	96	0	16.8	8.4	Fast WRIT_BUF
MLP	96	0	15.8	7.9	Fast WRIT_BUF
MPI	96	96	10.4	5.2	NUMTHREADS=2
MLP	96	96	10.2	5.1	OpenMP load-balanced
MPI	96	192	15.7	7.8	NUMTHREADS=3
MLP	96	192	10.4	5.2	OpenMP load-balanced

Table 2: CFL3D Complete Aircraft Timings on SGI 3000 Lomax

Type	Process Count	OpenMP Threads	Firstcycle	Finegrid	Comment
MPI	96	288	7.4	3.7	NUMTHREADS=4
MLP	96	288	8.6	4.3	OpenMP load-balanced

The insertion of the fast WRIT_BUF routine produced a speedup of about 3 relative to the original I/O for MPI multilevel versions with 32 high-level processors. Wallclock time for the MLP version executing with 32 high-level processors and with 0 OpenMP threads was twice as large as the corresponding MPI version and wallclock timings showed that the RESP timings were about factor of 2 greater than the corresponding MPI timings. Section 6 explained why the MLP load-barrier-store construction in subroutine RESP executes more slowly than the MPI Send/Recv construction.Doubling the number of threads involved in the main computational loop produced a speedup of 4.8 for the MPI version and a speedup of 2.3 for the MLP version when employing 32 OpenMP threads; however, the MLP wallclock time continued to exceed that of the MPI version on the calculation.
When employing 96 processors, the MPI version displayed a speedup of 4.1 relative to the original version. With an additional 96 OpenMP threads, the MPI version increases this speedup to 6.7. An additional set of 96 threads does not improve the speedup due to the loss of data locality. Setting NUMTHREADS to 3 violates the local data rule because, with 4 cpus per node, many OpenMP threads do not execute on the same node as their high-level processor, requiring data transfer over the network. With NUMTHREADS set to 4, the calculation recovers the data locality, displaying a speedup of 9.4
The MLP version performs better with 96 processors because the larger number of processes means fewer RESP passes per process and leads to a smaller contribution to the MLP wallclock time. Additional sets of 96 OpenMP threads produced speedup improvements similar to the MPI version.With 288 additional OpenMP threads the MLP version displays a speedup of 6.6.
The reason the MLP performance lags the MPI performance on LOMAX is the slower performance of the MLP load-barrier-store relative to the MPI Send/Recv in RESP.

Columbia

The Columbia system consists of multiple SGI Altix 512 CPU systems, each CPU having a peak rate of 6.0 GFLOPs with a memory of 2 GByte and a 6 Mbyte cache. Table 3 compares CFL3D performance on the Columbia system, illustrating the effects of the fast I/O and multi-level parallelism.Version 3.2 of the MLP library allows a set- and-test synchronization equivalent to the pairwise MPI Send/Receive. This functionality strongly influences the performance of the RESP routine which involves the master receiving data from the workers in order to evaluate and print the residuals.

Table 3: CFL3D Complete Aircraft Timings on SGI ALTIX Columbia

Type	Process Count	OpenMP Threads	Firstcycle	Finegrid	Comment
MPI	32	0	26.9	14.1	Original WRIT_BUF
MPI	32	0	14.3	7.2	Fast WRIT_BUF

Table 3: CFL3D Complete Aircraft Timings on SGI ALTIX Columbia

Type	Process Count	OpenMP Threads	Firstcycle	Finegrid	Comment
MLP	32	0	9.0	4.5	Fast WRIT_BUF
MPI	32	32	9.2	4.8	NUMTHREADS=2
MLP	32	32	5.6	2.9	OpenMP load-balanced
MPI	32	64	43.8	21.9	NUMTHREADS=3
MLP	32	64	4.8	2.4	OpenMP load-balanced
MPI	32	96	7.6	3.8	NUMTHREADS=4
MLP	32	96	5.8	2.9	OpenMP load-balanced
MPI	96	0	12.0	6.2	Original WRIT_BUF
MPI	96	0	7.5	3.7	Fast WRIT_BUF
MLP	96	0	5.3	2.7	Asynchronous RESP
MPI	96	96	4.2	2.1	NUMTHREADS=2
MLP	96	96	2.8	1.4	OpenMP load-balanced
MPI	96	192	12.7	6.5	NUMTHREADS=3
MLP	96	192	2.2	1.0	OpenMP load-balanced
MPI	96	288	3.8	1.9	NUMTHREADS=4
MLP	96	288	1.8	0.9	OpenMP load-balanced

Compared to the Lomax platform, the application of the fast WRIT_BUF on Columbia produced a speedup of about 2 relative to the original I/O for multilevel versions with 32 high-level processes and about 1.6 for the multilevel versions with 96 processors. Since MPI processes spend substantially less time in I/O on Columbia than on Lomax, the fast WRIT_BUF has less time to reduce on Columbia, so the speedup is smaller.

Adding 32 OpenMP threads to the MPI 32-process case increased the speedup to 2.9 relative to the

original I/O for multilevel versions with 32 high-level processes. As on Lomax, setting NUMTHREADS to 3 increased wallclock time as data locality is lost. A NUMTHREADS setting of 4 recovered the data locality and produced a speedup of 3.5 (26.9/7.6) relative to the original I/O for multilevel versions with 32 high-level processes. The MLP version with 0 OpenMP threads displays a speedup of 3.0 for 32 high-level processes; wallclock timings showed that the fast set-and-test routine in RESP was responsible for the greater speedup relative to the MPI version (3.0 vs. 2) improvement.With 32 OpenMP threads involved the main computational loop, the MLP speedup increased to 4.8.
When employing 96 OpenMP processes, the MPI version displays a speedup 1.6 and the MLP version displays a speedup of 2.3 relative to the original I/O for multilevel versions with 96 high-level processes. The MLP code is somewhat more effective at improving performance because MLP can assign OpenMP threads in proportion to the computational load assumed by the high-level process.Adding 288 threads to the original 96 processes increases the speedup to 3.2 for the MPI version and 6.7 for the MLP version; detailed timings show that the large grids are controlling the performance and they are unable to use effectively more than 10 threads.
Both Tables show that implementing a second-level of parallelism can add considerable value to the code as a design tool due to the faster turnaround. On Lomax, the official released version of CFL3D requires about 89 seconds on the finegrid calculation and the modifications discussed herein reduce this time to about 10 seconds. On Columbia, the official released version of CFL3D requires about 26.9 seconds on the finegrid calculation and the modifications discussed herein reduce this time to less than 2 seconds.

9. Conclusion

This report has described the creation of multilevel parallel versions of CFL3D and the subsequent performance gains displayed by those versions.The complexity of the code required an aggressive effort to achieve those gains, but the concentration of the CFL3D computational load in a single large loop simplified that effort. A careful reader will have encountered herein solutions to many of the problems attending the creation of a multilevel parallel code. Extensive use of these versions should amortize the creation overhead.
The major contributor to run-time improvement was the output optimization which was motivated by its role as the major bottleneck at all levels of parallelism. Additional OpenMP parallelism benefited both MPI and MLP versions of CFL3D with the MLP version demonstrating a somewhat greater performance improvement due to its load-balancing ability.
CFL3D was not designed to execute in a multilevel parallel mode and the fact that it does execute in a relatively efficient fashion indicates that this approach, i.e., the active creation of multilevel parallel regions, has considerable applicability, both to codes that have already been parallelized and codes that have yet to be parallelized.

References

[1] S.L.Krist, R.T. Biedron, and C.L. Rumsey, “CFL3D User’s Manual”, NASA/TM-1998-208444, June, 1998,
[2] J. R.Taft, “Performance of the OVERFLOW-MLP CFD Code on the NASA Ames 512 CPU Origin System,” In NASA HPCCP/CAS Workshop, NASA Ames Research Center, February 2000.
[3] http://www.OpenMP.org, OpenMP Fortran Application Program Interface.